Universidad de La Habana

Facultad de Matemática y Computación

Proyecto de Diseño de Análisis y Algoritmos

Análisis y resolución de ejercicios de la plataforma Codeforces

Rafael Acosta Márquez C-411
Eisler Francisco Valles Rodríguez C-411

Proyecto en github

Índice

1. AlmostSorted

1.1. Ejercicio

Link del problema en la plataforma **Codeforces**

- límite de tiempo por test: 2 segundos
- límite de memoria por test: 256 megabytes

Dada una permutación p de longitud n y un entero positivo k, el problema consiste en construir otra permutación q de los mismos elementos que minimice el número de inversiones bajo la restricción:

$$\forall\, 1 \leq i < j \leq n, \quad p_i \leq p_j + k. \tag{1}$$

Entrada

La primera línea contiene 2 enteros n, k $1 \leq n \leq 5000; 1 \leq k \leq 8$.
Luego siguen n enteros entre 1 y n que representan la permutación p.
Salida

La salida debe ser un único entero x que representa el menor número posible de inversiones para todos los posibles arreglos q que cumplen las restricciones planteadas en el problema.

1.2. Etiquetas

- DP
- Cubrimiento Mínimo

1.3. Solución

El algoritmo resuelve el problema mediante una combinación de programación dinámica (DP) y un Árbol de Fenwick (BIT). Se define un estado de la DP por una pareja `(block_start, mask)`, donde `block_start` indica el siguiente elemento (en orden creciente) que debe ser colocado en la permutación q, y `mask` es una máscara de bits de longitud $k+1$ que registra qué candidatos dentro de la ventana actual ya han sido seleccionados. Cada transición consiste en escoger un candidato disponible, actualizar el BIT para contabilizar las inversiones que se generan al colocar dicho elemento y reconfigurar la ventana mediante un desplazamiento, lo que permite explorar todas las soluciones válidas sin incurrir en la complejidad del enfoque por fuerza bruta.
Definiciones:

- **Permutación:** Un arreglo de n elementos donde cada entero de 1 a n aparece exactamente una vez.
- **Inversión:** Un par (i, j), $1 \leq i < j \leq n$, tal que $q_i > q_j$ en la permutación q.
- **Restricción del problema:** Para todo par $i < j$, se cumple que $p_i \leq p_j + k$, lo que impone un límite en el retraso relativo de los elementos en q respecto a su orden natural.
- **DP State (`block_start, mask`):** Representa el proceso de asignación de elementos, donde `mask` indica qué elementos dentro del bloque actual de $k+1$ candidatos ya han sido utilizados.
- **BIT (Fenwick Tree):** Estructura de datos que permite actualizar y consultar la cantidad de inversiones de forma eficiente en tiempo $O(\log n)$.

1.3.1. Fuerza bruta

Una solución de fuerza bruta implicaría enumerar todas las permutaciones posibles de q, verificar la restricción $p_i \leq p_j + k$ en cada caso y calcular el número de inversiones de cada permutación. Dado que existen $n!$ permutaciones, este enfoque tiene una complejidad factorial, lo cual es inviable para n tan grandes como 5000.

1.4. DP- Solution

La solución mediante programación dinámica explota el hecho de que, debido a la restricción impuesta por k, cada elemento en q puede provenir únicamente de una ventana de candidatos de tamaño acotado ($k+1$ elementos). Se definen los estados de la DP como `(block_start, mask)`, y para cada estado se exploran las transiciones posibles mediante la selección de un candidato no utilizado en la ventana actual. La función `dp` acumula el costo (en número de inversiones) sumando las inversiones locales, determinadas eficientemente mediante el BIT, y el costo futuro obtenido de los estados recursivos. La memoización evita el recálculo de estados ya evaluados, garantizando que la complejidad total sea $O(n \cdot 2^{(k+1)} \cdot k \cdot \log n)$, lo que resulta factible para los valores dados de n y k. Notar que se puede hacer el cálculo de las inversiones sin usar un Fenwick Tree, pero esto aumentaría la complejidad de la solución propuesta en un factor de N.

1.5. Demostración por Inducción

1.5.1. Paso Inductivo

Supongamos que, para todo estado con

$$\texttt{block\_start}' > \texttt{block\_start},$$

la función

$$\texttt{dp(block\_start}', \texttt{mask}')$$

calcula correctamente el mínimo número de inversiones acumulables a partir de dicho estado. Entonces, en el estado

$$\texttt{dp(block\_start, mask)},$$

al iterar sobre todas las posibles elecciones válidas de `idx` (es decir, todos los candidatos en la ventana de tamaño $k+1$ que aún no han sido usados), se evalúan:

- El **costo local** (número de inversiones producidas por la inserción del elemento elegido), obtenido mediante la consulta al BIT.
- El **costo futuro** (el mínimo costo de inversiones a partir del estado actualizado), garantizado por la hipótesis inductiva.

Se toma el mínimo sobre todas estas alternativas. Por lo tanto, la función

$$\texttt{dp(block\_start, mask)}$$

retorna el mínimo número de inversiones que se pueden obtener completando la construcción de q a partir de dicho estado.

1.6. Pertenecia a la clase NP

Considérese la versión de decisión del problema: dada una permutación p, dos enteros positivos n y k, y un umbral T, ¿existe una permutación q de $[1, n]$ que satisfaga la restricción

$$\forall\, 1 \le i < j \le n, \quad p_i \le p_j + k,$$

y que tenga a lo sumo T inversiones? Un certificado para esta decisión es la propia permutación q. Verificar que q es una permutación (es decir, que contiene cada entero de 1 a n exactamente una vez) se realiza en tiempo $O(n)$. Además, se puede comprobar en tiempo polinomial que la restricción se cumple (por ejemplo, en tiempo $O(n^2)$) y calcular el número de inversiones en tiempo $O(n \log n)$ utilizando una estructura tipo Árbol de Fenwick o simplemente en tiempo $O(n^2)$ mediante fuerza bruta. Por lo tanto, la versión de decisión del problema pertenece a NP.

1.7. Pertenencia a la clase NP-Hard

Para demostrar que el problema es NP-Hard, se realiza una reducción desde el problema de *Minimum Feedback Arc Set in Tournaments* (MFAS), conocido por ser NP-Hard.Charbit et al. [2006]

Instancia de MFAS: Dado un torneo $T = (V, A)$ con $|V| = n$, el problema MFAS consiste en encontrar un ordenamiento (permutación) q de los vértices que minimice el número de arcos invertidos, es decir, aquellos arcos (u, v) tales que, en el orden q, u aparece después de v.

Construcción de la instancia:

1. Sea p^* una extensión lineal (un ordenamiento Hamiltoniano) de T. Es bien sabido que, en todo torneo, existe un camino Hamiltoniano, y dicho ordenamiento puede obtenerse en tiempo polinomial.
2. Se define la permutación de entrada p como $p = p^*$.
3. Se fija $k = n - 1$. Con esta elección, la restricción

$$p_i \leq p_j + k$$

se satisface trivialmente para toda permutación q, ya que para cualquier i se tiene $p_i \leq n$ y $p_j + (n-1) \geq n$.

Relación entre la inversión y el feedback:
Para cualquier permutación q de los vértices, definimos el número de inversiones relativo a p como

$$\text{inv}(q) = \#\{(i, j) \mid 1 \leq i < j \leq n \text{ y } p^{-1}(q(i)) > p^{-1}(q(j))\}.$$

Dado que $p = p^*$ es una extensión lineal de T, para cada par de vértices (u, v) se cumple:

- Si T contiene el arco (u, v) (es decir, u es preferido a v), entonces en p se tiene $p(u) < p(v)$.
- Por lo tanto, si en el ordenamiento q se invierte la relación (es decir, v aparece antes que u), se incurre en una inversión que refleja el incumplimiento de la preferencia impuesta por T.

Más formalmente, se puede demostrar que

$$\text{inv}(q) = \text{FAS}(q) + C,$$

donde $\text{FAS}(q)$ es el número de arcos invertidos (o feedback arcs) en T respecto al ordenamiento q, y C es una constante dependiente únicamente de la instancia fija (determinada por p^* y T). En consecuencia, minimizar $\text{inv}(q)$ es equivalente a minimizar $\text{FAS}(q)$.

Conclusión de la Reducción:
Dado que MFAS es NP-Hard en torneos, y dado que la reducción descrita transforma cualquier instancia de MFAS en una instancia de nuestro problema (donde la restricción es trivial al fijar $k = n-1$), se concluye que el problema de encontrar una permutación q que minimice el número de inversiones relativo a p es NP-Hard cuando n y k son variables de la entrada.

Nota: Aunque en esta reducción se elige $k = n - 1$ para hacer trivial la restricción, dado que k es parte de la entrada, la NP-Hardness se extiende a la versión general del problema en la que k puede tomar cualquier valor.

2. Complete the Projects (hard version)

2.1. Ejercicio

Link del problema en la plataforma **Codeforces**

- límite de tiempo por test: 2 segundos
- límite de memoria por test: 256 megabytes

Piad es un científico de la computación muy famoso. Su calificación actual es de r unidades.

Algunos clientes muy ricos le pidieron que completara algunos proyectos para sus empresas. Para realizar el i proyecto, Piad necesita tener al menos ai unidades de calificación; después de completar este proyecto, su calificación cambiará en bi (su calificación aumentará o disminuirá en bi) (bi puede ser positivo o negativo). La calificación de Piad no debe caer por debajo de cero porque entonces la gente no confiará en un científico de la computación tan poco valorado.

Su tarea consiste en calcular el tamaño máximo posible de dicho subconjunto de proyectos.

- 1 — Piad puede elegir el orden en el que completa los proyectos. Además, puede incluso saltarse algunos proyectos.
- 2 — Para ganar más experiencia (y dinero, por supuesto) Piad quiere elegir el subconjunto de proyectos que tengan el máximo tamaño posible y el orden en el que los completará, de forma que tenga suficiente calificación antes de empezar cada proyecto, y tenga una calificació no negativa después de completar cada proyecto.

Entrada

La primera línea contiene dos enteros n y r ($1 \leq n \leq 100, 1 \leq r \leq 30000$).

Las siguientes n líneas contienen proyectos, uno por línea. El i-ésimo proyecto es representado como un par de enteros a_i y b_i ($1 \leq a_i \leq 30000, -300 \leq b_i \leq 300$) - la valoración requerida para completar el i-ésimo proyecto y el cambio de calificación tras la finalización del proyecto

Salida

Imprime un número entero: el tamaño del subconjunto máximo posible (posiblemente, vacío) de proyectos que Piad puede elegir.

2.2. Etiquetas

- DP
- Greedy

2.3. Solución

Para abordar el problema se observa que los proyectos se pueden clasificar en dos tipos:

- **Proyectos positivos** ($b_i \geq 0$): Se pueden realizar en cuanto estén disponibles, ya que aumentan o mantienen la calificación.
- **Proyectos negativos** ($b_i < 0$): Se deben realizar de forma cuidadosa, ya que reducen la calificación.

Se presentan dos enfoques: uno basado en **programación dinámica (DP)** y otro en un algoritmo **greedy (voraz)**.

Definiciones y Formulación del Problema DP

Sea $\mathcal{P} = \{(a_i, b_i) \mid b_i < 0\}$ el conjunto de proyectos negativos. Se ordena $\mathcal{P}$ de forma que, para dos proyectos $p = (a_p, b_p)$ y $q = (a_q, b_q)$, si p aparece antes que q se cumple

$$a_p + b_p \geq a_q + b_q.$$

Esta ordenación "prioriza" aquellos proyectos que, pese a su efecto negativo, dejan una mayor calificación residual tras su realización.

Definimos el arreglo $\{dp[j]\}_{j=0}^{m}$ (con $m = |\mathcal{P}|$) de la siguiente forma:

$dp[j]$ = máxima calificación alcanzable tras ejecutar exactamente j proyectos negativos en el orden establecido,

con la restricción de que la calificación se mantenga no negativa en todo momento. Se inicializa:

$$dp[0] = r,$$

donde r es la calificación tras procesar los proyectos positivos, y para $j \geq 1$ se define inicialmente

$$dp[j] = -\infty.$$

La transición se formula de la siguiente manera: Para cada proyecto negativo (a, b) (con $b < 0$) y para $i = m-1, m-2, \ldots, 0$, si se tiene

$$dp[i] \geq a \quad \text{y} \quad dp[i] + b \geq 0,$$

entonces se actualiza

$$dp[i+1] = \text{máx}\{dp[i+1],\, dp[i] + b\}.$$

2.3.1. Programación Dinámica

Procesamiento de Proyectos Positivos

1. Se ordenan los proyectos positivos por a_i en orden ascendente.
2. Se ejecutan secuencialmente comprobando que la calificación actual r sea al menos a_i. Si se cumple la condición, se realiza el proyecto y se actualiza la calificación:

$$r \leftarrow r + b_i.$$

3. Se cuenta el número de proyectos positivos completados, denotado por `countPositive`.

Procesamiento de Proyectos Negativos mediante DP

Sea $\mathcal{P} = \{(a_i, b_i) \mid b_i < 0\}$ el conjunto de proyectos negativos. Se ordena $\mathcal{P}$ de forma que, para dos proyectos $p = (a_p, b_p)$ y $q = (a_q, b_q)$, si p aparece antes que q se cumple

$$a_p + b_p \geq a_q + b_q.$$

Esta ordenación "prioriza" aquellos proyectos que, pese a su efecto negativo, dejan una mayor calificación residual tras su realización.

Definimos el arreglo $\{dp[j]\}_{j=0}^{m}$ (con $m = |\mathcal{P}|$) de la siguiente forma:

$dp[j]$ = máxima calificación alcanzable tras ejecutar exactamente j proyectos negativos en el orden establecido,

con la restricción de que la calificación se mantenga no negativa en todo momento. Se inicializa:

$$dp[0] = r,$$

donde r es la calificación tras procesar los proyectos positivos, y para $j \geq 1$ se define inicialmente

$$dp[j] = -\infty.$$

La transición se formula de la siguiente manera: Para cada proyecto negativo (a, b) (con $b < 0$) y para $i = m-1, m-2, \ldots, 0$, si se tiene

$$dp[i] \geq a \quad \text{y} \quad dp[i] + b \geq 0,$$

entonces se actualiza

$$dp[i+1] = \text{máx}\{dp[i+1],\, dp[i] + b\}.$$

Se define:

$$\texttt{bestNegative} = \text{máx}\{\, j \mid dp[j] \geq 0 \,\}.$$

La solución final es la suma de los proyectos positivos y negativos completados:

$$\texttt{countPositive} + \texttt{bestNegative}.$$

Demostración de Correctitud

La demostración se divide en dos partes: (1) se prueba que es posible, sin pérdida de generalidad, considerar únicamente órdenes en las que los proyectos negativos aparecen en forma no creciente según $a_i + b_i$; y (2) se demuestra, por inducción, que la recurrencia DP definida produce la máxima calificación alcanzable tras ejecutar j proyectos.

Paso 1: Validez de la Ordenación por $a_i + b_i$

Lema 1. *Sean $p = (a_p, b_p)$ y $q = (a_q, b_q)$ dos proyectos negativos tales que*

$$a_p + b_p < a_q + b_q.$$

Si en una secuencia factible S se ejecuta p antes que q, entonces existe una secuencia factible S' en la que se intercambian p y q (es decir, primero se ejecuta q y luego p).

Demostración. Sea R la calificación inmediatamente antes de ejecutar p en la secuencia S. Dado que S es factible se tiene:

$$R \geq a_p, \quad \text{(para poder ejecutar } p\text{)} \tag{2}$$

$$R + b_p \geq a_q, \quad \text{(para ejecutar } q \text{ tras } p\text{)} \tag{3}$$

$$R + b_p + b_q \geq 0. \quad \text{(calificación final no negativa)} \tag{4}$$

Ahora, consideremos la secuencia S' en la que se ejecuta primero q y luego p. Sea también R la calificación inicial para q en S'. Para que S' sea factible se deben cumplir:

$$R \geq a_q, \tag{5}$$

$$R + b_q \geq a_p, \tag{6}$$

$$R + b_q + b_p \geq 0. \tag{7}$$

Observemos lo siguiente:

- **Condición (5):** De (3) se tiene $R \geq a_q - b_p$. Dado que $b_p < 0$, se tiene $-b_p > 0$ y, por tanto, $a_q - b_p > a_q$. Así, $R \geq a_q - b_p$ implica en particular que $R \geq a_q$.
- **Condición (6):** Queremos demostrar que $R + b_q \geq a_p$. Notemos que la hipótesis del lema, $a_p + b_p < a_q + b_q$, se reescribe como:

  $$a_q + b_q > a_p + b_p \quad \Longrightarrow \quad a_q + b_q - b_p > a_p.$$

  Dado que, en S, se cumple (3): $R + b_p \geq a_q$, sumando $b_q - b_p$ a ambos lados se obtiene

  $$R + b_q \geq a_q + b_q - b_p > a_p.$$

- **Condición (7):** Es idéntica a la (4) (recordando que la suma $b_p + b_q$ es conmutativa).

Por lo tanto, la secuencia S' es factible. Repetidamente, mediante intercambios locales, cualquier secuencia factible puede transformarse en una en la que los proyectos negativos se ejecuten en orden no creciente según $a_i + b_i$.

□

Paso 2: Correctitud de la Recurrencia DP

Procederemos por inducción sobre el número j de proyectos negativos ejecutados.

Caso base: Para $j = 0$ no se ha ejecutado ningún proyecto negativo, de modo que la calificación final es

$$dp[0] = r,$$

lo cual es óptimo.

Paso inductivo: Sea $j \geq 0$ y supongamos que $dp[j]$ es la máxima calificación alcanzable tras ejecutar j proyectos negativos en el orden no creciente de $a_i + b_i$. Sea $p = (a, b)$ el siguiente proyecto (el de posición $j+1$ en el orden) y consideremos la posibilidad de ejecutar p inmediatamente después de haber alcanzado la calificación $dp[j]$. Para que p sea ejecutable es necesario que:

$$dp[j] \geq a \quad \text{y} \quad dp[j] + b \geq 0.$$

En ese caso, se puede actualizar:

$$dp[j+1] = \text{máx}\{dp[j+1],\, dp[j] + b\}.$$

Por hipótesis, cualquier secuencia factible de $j+1$ proyectos no puede dejar una calificación mayor que $dp[j] + b$; de allí se deduce que $dp[j+1]$ es óptimo.

Corolario 1. *El número máximo de proyectos negativos que se pueden ejecutar de forma factible es*

$$k = \text{máx}\{\, j \mid dp[j] \geq 0 \,\}.$$

Análisis de Complejidad

Procesamiento de Proyectos Positivos

- **Ordenamiento:** Los proyectos positivos se ordenan en orden creciente según a_i. Dado que existen n proyectos positivos, esta operación requiere $\mathcal{O}(n \log n)$.
- **Ejecución:** Se recorre la lista de proyectos positivos y, para cada proyecto, se verifica la condición $r \geq a_i$ y se actualiza la calificación r (más una operación de incremento). Esto se realiza en $\mathcal{O}(n)$.

Por lo tanto, la complejidad total para la fase de proyectos positivos es:

$$\mathcal{O}(n \log n).$$

Procesamiento de Proyectos
Sea m el número de proyectos negativos. La fase DP consta de los siguientes pasos:

1. **Ordenamiento:** Los proyectos negativos se ordenan en orden decreciente según $a_i + b_i$. Esta operación requiere $\mathcal{O}(m \log m)$.
2. **Inicialización del Arreglo DP:** Se define un arreglo $dp[0 \ldots m]$ donde:

   $$dp[0] = r \quad \text{y} \quad dp[j] = -\infty \quad \text{para } j \geq 1.$$

   Esta inicialización requiere $\mathcal{O}(m)$.
3. **Actualización mediante Doble Bucle:** Para cada proyecto negativo (a, b) (total de m proyectos), se recorre el arreglo dp desde $i = m - 1$ hasta 0. En cada iteración se verifica si:

   $$dp[i] \geq a \quad \text{y} \quad dp[i] + b \geq 0,$$

   y en caso afirmativo se actualiza:

   $$dp[i+1] = \text{máx}\{dp[i+1],\, dp[i] + b\}.$$

   Dado que, en el peor caso, el bucle interno itera m veces para cada uno de los m proyectos negativos, esta parte requiere:

   $$\mathcal{O}(m^2).$$

Complejidad Global
Sumando las complejidades de ambas fases, se tiene:

$$\mathcal{O}(n \log n + m \log m + m + m^2).$$

Observando que el término dominante es $\mathcal{O}(m^2)$ y que, en el peor caso, m puede ser del mismo orden que n (ya que $n = n + m$ y $n \leq 100$ según las restricciones del problema), se concluye que la complejidad global del algoritmo de programación dinámica es:

$$\boxed{\mathcal{O}(n^2) \quad \text{en el peor de los casos.}}$$

2.3.2. Greedy

Procesamiento de Proyectos Positivos

Procedimiento del Algoritmo La función `MaxPos` realiza los siguientes pasos:

a) Se toma el conjunto
$$P = \{(a_i, b_i) : b_i \geq 0\},$$
y se ordena de forma creciente según a_i (el rating mínimo requerido).

b) Se recorre la lista en ese orden. Para cada proyecto (a_i, b_i) se verifica que el rating actual r cumpla $r \geq a_i$. En ese caso, se realiza.[el] proyecto, aumentando r en b_i y se incrementa el contador global de proyectos realizados.

Demostración de Optimalidad Afirmación: Si existe un orden de realización de los proyectos de P que permite completarlos sin violar las condiciones (tener rating suficiente al iniciar y no caer por debajo de 0 tras su ejecución), entonces el procedimiento *greedy* —que toma los proyectos en orden creciente de a_i— completa al menos la misma cantidad de proyectos.

Argumento (por intercambio e inducción):

- *Caso base:* Si P es vacío o contiene un único proyecto, el algoritmo es trivialmente óptimo.
- *Paso inductivo:* Sea π una secuencia óptima de proyectos de P y supongamos que en π aparece un proyecto $P_t = (a_t, b_t)$ realizado en una posición posterior respecto a otro proyecto $P_s = (a_s, b_s)$ con $a_s < a_t$. Dado que $b_s \geq 0$, realizar P_s antes de P_t incrementa (o al menos no disminuye) el rating, lo que facilita la realización de P_t. Por un argumento de intercambio se concluye que cualquier secuencia óptima puede reordenarse de forma creciente respecto a a_i sin perder factibilidad ni reducir el número de proyectos completados.

De esta forma, el algoritmo que recorre P en orden creciente de a_i es óptimo para la parte de los proyectos positivos.

Procesamiento de Proyectos Negativos

Planteamiento del Subproblema Sea R el rating resultante tras completar todos los proyectos positivos (es decir, el valor final de r después de ejecutar `MaxPos`). Los proyectos negativos son aquellos para los cuales $b_i < 0$. Para cada proyecto, se deben satisfacer dos condiciones:

1. **Requisito previo:** Al iniciar el proyecto se debe tener rating al menos a_i. Sea d la caída acumulada (tal que el rating actual es $R - d$); entonces se requiere:
$$R - d \geq a_i \quad \Longleftrightarrow \quad d \leq R - a_i.$$
2. **No caer en negativo:** Tras ejecutar el proyecto y sumar b_i (con $b_i < 0$), se debe tener:
$$R - d + b_i \geq 0 \quad \Longleftrightarrow \quad d \leq R + b_i.$$

Dado que $b_i < 0$, la inecuación $d \leq R + b_i$ es más restrictiva que $d \leq R$. En resumen, para que el proyecto i sea realizable cuando la caída acumulada es d, es necesario que
$$d \leq \text{mín}\{R - a_i,\ R + b_i\}.$$

Observamos que, en una secuencia de proyectos negativos, d es la suma de los *costos* de cada proyecto, donde se define el costo como
$$c_i = -b_i > 0.$$

Transformación en el Algoritmo En el código se transforma cada proyecto negativo, originalmente representado por el par (a_i, b_i), modificando su primer componente de la siguiente forma:

a) Se calcula
$$p.X \leftarrow R - a_i.$$

b) Se actualiza restando b_i (recordando que $b_i < 0$, esta resta equivale a sumar $c_i = -b_i$):

$$p.X \leftarrow (R - a_i) - b_i = R - a_i - b_i.$$

c) Finalmente, se fija:

$$p.X \leftarrow \text{mín}\{p.X,\, R\}.$$

Definimos, para cada proyecto negativo, el valor

$$K_i = \text{mín}\{R,\, R - a_i - b_i\}.$$

Analicemos los casos:

- Si $a_i > -b_i$ (es decir, $a_i > c_i$), entonces $R - a_i < R + b_i$ y la restricción de factibilidad es $d \leq R - a_i$. En este caso, $K_i = R - a_i - b_i = (R - a_i) + c_i$.
- Si $a_i \leq -b_i$, se tiene $R + b_i \leq R - a_i$ y la restricción es $d \leq R + b_i$. Además, como $R - a_i - b_i \geq R$, se fija $K_i = R$.

Así, la transformación asocia a cada proyecto negativo un par (K_i, c_i), donde $c_i = -b_i$ y $K_i \in [0, R]$. Aunque K_i no es idéntico a la restricción original $D_i = \text{mín}\{R - a_i,\, R + b_i\}$, la forma en que se utiliza en el algoritmo permite agrupar los proyectos según un parámetro que facilita la asignación del "presupuesto"de rating (o, en la analogía, del tiempo disponible).

El Algoritmo de `Schedule` La función `Schedule` procede de la siguiente forma:

1. Se ordenan los proyectos negativos (ya transformados) en orden creciente según K_i.
2. Se recorre la lista en orden inverso (de mayor a menor K_i). Para cada proyecto i se define:
   - `current` $= \text{máx}\{0,\, K_i\}$.
   - Si existe un proyecto previo (en la lista ordenada), se define:

     $$\texttt{previous} = \text{máx}\{0,\, K_{i-1}\},$$

     y se toma la diferencia:

     $$\Delta = \texttt{current} - \texttt{previous}.$$

     (Si i es el primero, se define `previous` $= 0$.)
3. En cada iteración se inserta en un multiconjunto el costo $c_i = -b_i$ del proyecto actual. Luego, mientras exista presupuesto $\Delta > 0$ y el multiconjunto no esté vacío, se extrae del multiconjunto el costo mínimo x y se procede de la siguiente forma:
   - Si $\Delta \geq x$, se "programa" (realiza) ese proyecto, se incrementa el contador global y se reduce Δ en x.
   - Si $\Delta < x$, se paga parcialmente x (reduciéndolo en Δ), se reinserta el costo remanente en el multiconjunto y se agota Δ.

Interpretación: El algoritmo divide el intervalo $[0, R]$ en subintervalos determinados por los valores consecutivos de K_i y, en cada ventana de tamaño Δ, asigna parte del presupuesto disponible para "pagar" el costo de los proyectos. Este procedimiento es análogo a los algoritmos *greedy* clásicos para maximizar el número de tareas que pueden programarse bajo restricciones de tiempos de procesamiento y deadlines, utilizando una cola de prioridad (en este caso, el `MultiSet`).

Demostración de Optimalidad (por Intercambio y Reducción a Scheduling) La idea central es la siguiente: Sea d la caída acumulada (o carga) que se va sumando al realizar proyectos negativos. Para cada proyecto negativo i con parámetros (a_i, b_i) (y costo $c_i = -b_i$), la factibilidad exige que, al iniciar el proyecto,

$$d \leq \text{mín}\{R - a_i,\, R + b_i\}.$$

Definimos:

$$D_i = \text{mín}\{R - a_i,\, R + b_i\}.$$

Una solución factible es una secuencia de proyectos negativos tal que, si d es la suma acumulada de los c_i en el orden de realización, se cumple que para cada proyecto realizado:

$$d \leq D_i.$$

Este problema se reduce a una versión del clásico problema de *scheduling* de tareas (con tiempos de procesamiento c_i) sujetas a deadlines D_i. Se sabe, por teoremas clásicos sobre algoritmos *greedy* para scheduling, que el siguiente procedimiento es óptimo:

a) Ordenar las tareas (en este caso, los proyectos negativos) según sus deadlines o parámetros relacionados.

b) Insertar las tareas en una cola de prioridad y, conforme se va "avanzando" en el tiempo (o en la acumulación del costo d), elegir siempre la tarea de menor costo para aprovechar al máximo el presupuesto disponible.

El algoritmo presentado transforma cada proyecto en un par (K_i, c_i) y asigna el presupuesto disponible en ventanas determinadas por los valores consecutivos de K_i. Un argumento por intercambio (similar al usado en el análisis del algoritmo de Moore o en otros algoritmos *greedy* de scheduling) permite concluir que, si existiera una solución que realizara un mayor número de proyectos negativos sin violar las restricciones, se podría reordenar dicha solución para que consumiera el presupuesto de manera idéntica al algoritmo.

Por lo tanto, el procedimiento implementado en `Schedule` es óptimo para seleccionar y ordenar los proyectos negativos de modo que el rating nunca caiga por debajo de 0.

Sea $S = \{x_1, x_2, \ldots, x_n\}$ un arreglo arbitrario de números (no necesariamente ordenado). Se desea demostrar que si existiese un algoritmo A que resuelve el problema de selección de proyectos en tiempo $o(n \log n)$, entonces se podría ordenar S en tiempo $o(n \log n)$. Dado que el problema de ordenamiento en el modelo de comparación tiene un límite inferior de $\Omega(n \log n)$, se llegaría a una contradicción, lo que implica que *no puede existir un algoritmo óptimo para el problema de selección de proyectos con complejidad menor a* $O(n \log n)$.

Complejidad

Teorema 1. *El algoritmo para seleccionar el máximo número de proyectos que Piad puede realizar tiene complejidad temporal* $O(n \log n)$ *en el peor caso.*

Demostración. Analicemos cada una de las fases del algoritmo:

1. **Lectura y Clasificación:** Se recorren n proyectos, clasificándolos en dos listas. El costo es:

$$O(n).$$

2. **Procesamiento de Proyectos Positivos (`MaxPos`):**

   - La ordenación de la lista positive tiene costo $O(n_{\text{pos}} \log n_{\text{pos}})$, con $n_{\text{pos}} \leq n$.
   - La iteración sobre la lista ordenada es $O(n_{\text{pos}})$.

   Por lo tanto, el costo total en esta fase es:

$$O(n \log n) + O(n) = O(n \log n).$$

3. **Modificación y Ordenación de Proyectos Negativos:** Se actualiza cada proyecto negativo en $O(1)$ y se ordena la lista de tamaño n_{neg}. Así, el costo es:

$$O(n_{\text{neg}}) + O(n_{\text{neg}} \log n_{\text{neg}}) = O(n \log n).$$

4. **Planificación de Proyectos Negativos (`Schedule`):** Se itera la lista negative (de tamaño n_{neg}) en orden inverso. En cada iteración se realizan:

   - Una inserción en la estructura multiconjunto: $O(\log n)$.
   - Un ciclo `while` que, en el peor caso, realiza un número limitado de operaciones (cada una de $O(\log n)$) y, de forma amortizada, cada elemento se procesa una única vez.

Por lo tanto, el costo total en esta fase es:

$$O(n_{\text{neg}} \log n) = O(n \log n).$$

Finalmente, sumando los costos de todas las fases se tiene:

$$O(n) + O(n \log n) + O(n \log n) + O(n \log n) = O(n \log n).$$

Por lo tanto, la complejidad temporal total del algoritmo es $O(n \log n)$ en el peor caso. □

2.4. Demostración de que $O(n \log n)$ es la mejor cota

Sea $S = \{x_1, x_2, \ldots, x_n\}$ un arreglo arbitrario de números (no necesariamente ordenado). Se desea demostrar que si existiese un algoritmo A que resuelve el problema de selección de proyectos en tiempo menor a $o(n \log n)$, entonces se podría ordenar S en tiempo menor a $o(n \log n)$. Dado que el problema de ordenamiento en el modelo de comparación tiene un límite inferior de $\Omega(n \log n)$, se llegaría a una contradicción, lo que implica que *no puede existir un algoritmo óptimo para el problema de selección de proyectos con complejidad menor a* $O(n \log n)$.

2.4.1. Construcción de la Reducción

Dado $S = \{x_1, x_2, \ldots, x_n\}$, procedemos de la siguiente manera:

1. Se define el rating inicial r como el mínimo elemento del arreglo:

$$r = \text{mín}\{x_1, x_2, \ldots, x_n\}.$$

2. Para cada $x_i \in S$ se crea una tarea (proyecto) P_i con:

$$a_i = x_i.$$

3. Se define la ganancia b_i de cada tarea de modo que, al completar las tareas en orden estrictamente creciente, el rating se incremente de forma exacta para alcanzar el siguiente umbral. Es decir, sea $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$ la secuencia de elementos de S en orden creciente (la reducción se diseña de modo que *la única solución óptima* para el problema de selección de proyectos es realizar las tareas en este orden). Se define:

$$b_{(1)} = 0, \quad \text{y para } i \geq 2, \quad b_{(i)} = x_{(i)} - x_{(i-1)}.$$

De esta forma, si se realizan las tareas en el orden $P_{(1)}, P_{(2)}, \ldots, P_{(n)}$, el rating evoluciona de la siguiente manera:

$$r_1 = r, \quad r_2 = r_1 + b_{(2)} = x_{(1)} + \big(x_{(2)} - x_{(1)}\big) = x_{(2)}, \quad \ldots, \quad r_n = x_{(n)}.$$

2.4.2. Argumento de Correctitud y Contradicción

Note que, con esta construcción:

- La única forma de que la secuencia de tareas sea factible (es decir, que para cada tarea se cumpla que el rating antes de la tarea es al menos a_i) es realizar las tareas en orden no decreciente de a_i, es decir, en orden creciente de los x_i.
- Por lo tanto, la solución óptima para el problema de selección de proyectos corresponde a la secuencia $\{P_{(1)}, P_{(2)}, \ldots, P_{(n)}\}$, la cual ordena implícitamente el arreglo S.

Ahora, supongamos que existe un algoritmo A que resuelve el problema de selección de proyectos en tiempo menor a $o(n \log n)$. Aplicando A a la instancia construida obtenemos en tiempo menor a $o(n \log n)$ la secuencia óptima de tareas, es decir, el arreglo S ordenado en forma creciente.

Sin embargo, es conocido que cualquier algoritmo de ordenamiento basado en comparaciones tiene un límite inferior de $\Omega(n \log n)$ en el peor caso. Por lo tanto, obtener una solución de ordenamiento en tiempo menor a $o(n \log n)$ es imposible. Se llega así a una contradicción.

Q.E.D

3. Button Lock

3.1. Ejercicio

Enunciado:

Estás parado frente a una habitación con grandes tesoros. Lo único que te detiene es una puerta con una cerradura de combinación de botones. Esta cerradura tiene d botones con dígitos del 0 a $d-1$. Cuando presionas un botón, este permanece presionado. No puedes desactivar un solo botón, pero existe un botón **RESET** que, al presionarlo, desactiva todos los botones. Inicialmente, ningún botón está presionado.

La puerta se abre instantáneamente cuando se presiona un conjunto específico de dígitos. Sin embargo, no conoces la contraseña; se sabe, por la documentación, que existen n posibles contraseñas. Se requiere encontrar la secuencia más corta de pulsaciones de botones (donde cada pulsación es un dígito o un RESET, indicado con la letra `R`) de modo que, durante su ejecución, aparezca al menos una vez cada una de las n contraseñas posibles.

Entrada:

- La primera línea contiene dos enteros d y n ($1 \leq d \leq 10$; $1 \leq n \leq 2^d - 1$).
- Las siguientes n líneas contienen cada una una cadena de d ceros y unos, en donde el j-ésimo carácter es 1 si y solo si el botón $j-1$ debe estar presionado.

Salida:

En la primera línea se debe imprimir el número mínimo k de pulsaciones; en la segunda, la secuencia de operaciones (los dígitos y las letras `R` para RESET).

Link del problema en Codeforces

3.2. Etiquetas

- Flujo de costo mínimo
- Matching Bipartito
- Programación Dinámica
- Bitmask

3.3. Solución

El algoritmo se basa en la siguiente observación:

Operaciones de la cerradura:

- **Pulsar un botón:** pasa el estado actual X a $X \cup \{i\}$. En un segmento sin RESET, se acumulan botones, y los estados que aparecen son los *prefijos* de la secuencia de pulsaciones.
- **RESET:** reinicia el estado a $\emptyset$.

Una *contraseña* S es un subconjunto de los d botones (representado mediante su bitmask) y tiene $|S|$ pulsaciones asociadas. Si cada contraseña se cubriera en un segmento separado, el costo total sería

$$C_{\text{sep}} = \sum_{i=1}^{n} |S_i| + (n-1),$$

donde $(n-1)$ corresponde a los RESET necesarios entre segmentos (no se requiere RESET antes del primer segmento).

Cuando se unen varias contraseñas en un mismo segmento (lo que es posible siempre que se verifique $S_i \subset S_j$ para contraseñas que aparezcan en la misma cadena) se puede *ahorrar* pulsaciones y RESET. En efecto, si en un segmento se tienen S_i y S_j con $S_i \subset S_j$, se cubren ambas pulsaciones con solamente $|S_j|$ pulsaciones en vez de $|S_i| + |S_j| + 1$ (considerando además el RESET entre segmentos). El ahorro al unir S_i con S_j es:

$$\Delta(i,j) = |S_i| + 1.$$

Sea B el beneficio total al unir contraseñas en cadenas. Entonces, el costo de la solución se puede expresar como:

$$C = \left(\sum_{i=1}^{n} |S_i| + (n-1) \right) - B.$$

Minimizar C equivale a maximizar B.

3.3.1. Reducción a Flujo de Costo Mínimo

Para lograr maximizar el beneficio B, se construye un grafo bipartito $G = (U \cup V, E)$ de la siguiente forma:

- Cada contraseña S_i se replica en ambos conjuntos: $i \in U$ y $i \in V$.
- Se añade una arista de i (en U) a j (en V) si y solo si $S_i \subset S_j$ (asegurando, además, que $|S_i| < |S_j|$).
- A cada arista (i, j) se le asigna **capacidad** 1 y **costo**

  $$c(i, j) = -\big(|S_i| + 1\big).$$

  De esta forma, enviar flujo a través de la arista representa unir S_i y S_j en la misma cadena, obteniéndose un beneficio de $|S_i| + 1$.

Además se introduce:

- Una *fuente* conectada a todos los nodos de U (capacidad 1, costo 0).
- Un *sumidero* al que se conectan todos los nodos de V (capacidad 1, costo 0).

Con esta construcción, un flujo unitario a través de la arista (i, j) indica que las contraseñas S_i y S_j se unirán en una misma cadena. Sea M el matching (conjunto de aristas usadas) y defínase el beneficio total obtenido como:

$$B(M) = \sum_{(i,j) \in M} \Big(|S_i| + 1 \Big).$$

Luego, el costo de la solución es:

$$C = \left(\sum_{i=1}^{n} |S_i| + (n-1) \right) - B(M).$$

El algoritmo de flujo de costo mínimo se encarga de encontrar el matching M^* que maximiza $B(M)$ (al minimizar el costo total del flujo), lo cual implica que la secuencia de operaciones reconstruida a partir de M^* es óptima.

3.4. Demostración de Correctitud

Rafael Bk, [2/9/2025 10:25 PM]

Teorema 2. *Sea $T = \{S_1, S_2, \ldots, S_n\}$ el conjunto de contraseñas (cada una representada mediante su bitmask) y sea*

$$C_{sep} = \sum_{i=1}^{n} |S_i| + (n-1)$$

el costo de cubrir cada contraseña en un segmento independiente. Entonces, la secuencia de operaciones generada mediante la reducción a flujo de costo mínimo, que produce un matching M con beneficio

$$B(M) = \sum_{(i,j) \in M} (|S_i| + 1),$$

tiene costo

$$C = C_{sep} - B(M),$$

el cual es mínimo. En consecuencia, la secuencia construida es óptima.

Demostración. Observemos lo siguiente:

- Si cada contraseña se cubriera en un segmento separado, el costo total sería C_{sep}.
- Si se unen dos contraseñas S_i y S_j (con $S_i \subset S_j$) en un mismo segmento, la cobertura de ambas se realiza al alcanzar S_j (con costo $|S_j|$ en vez de $|S_i| + |S_j| + 1$), obteniéndose un ahorro de

$$\Delta(i, j) = |S_i| + 1.$$

Si una cadena C contiene k contraseñas, el ahorro total es

$$B_C = \sum_{t=1}^{k-1} \Big(|S_{i_t}| + 1 \Big).$$

Dividiendo T en m cadenas, el costo total de la solución es:

$$C = \sum_{C \text{ cadena}} |S_{\text{máx}}(C)| + (m - 1).$$

Notamos que este costo se puede reescribir como:

$$C = C_{\text{sep}} - B,$$

donde B es el ahorro total acumulado al unir contraseñas en cadenas.

En el grafo bipartito construido, cada arista (i, j) tiene costo $-(|S_i| + 1)$. Así, si se envía un flujo unitario a través de dicha arista, se aporta un "beneficio"de $|S_i|+1$. Sea $B(M)$ la suma de estos beneficios para el matching M obtenido. Entonces, el costo total del flujo es:

$$C_M = -B(M).$$

El algoritmo de flujo de costo mínimo encuentra el matching M^* que maximiza $B(M)$ (o, equivalentemente, minimiza C_M). Así, la solución final tiene costo:

$$C = \left(\sum_{i=1}^{n} |S_i| + (n - 1) \right) - B(M^*),$$

el cual es mínimo.

La reconstrucción de la secuencia se efectúa de la siguiente manera:

1. Cada arista (i, j) en el matching M^* indica que S_i y S_j serán cubiertas en la misma cadena.
2. Los nodos sin arista entrante constituyen el inicio de una cadena.
3. Para cada cadena, partiendo del estado $\emptyset$, se presionan los botones necesarios hasta alcanzar el estado final de la cadena. De este modo, los estados intermedios (que corresponden a las contraseñas de la cadena) aparecerán en algún punto de la secuencia.
4. Se inserta la operación RESET (`R`) entre cadenas para reiniciar el estado.

De ello se concluye que la secuencia de operaciones generada es de costo mínimo y, por tanto, óptima. □

3.5. Complejidad Temporal

Teorema 3. *El algoritmo propuesto para resolver el problema de la cerradura de combinación, que utiliza una reducción a flujo de costo mínimo en un grafo bipartito, tiene una complejidad temporal en el peor caso de*

$$O(n^4),$$

donde n es el número de contraseñas posibles.

Demostración. Analizaremos la complejidad del algoritmo dividiéndolo en tres fases principales:

1. Construcción del grafo:

- Se tiene un conjunto U con n nodos y un conjunto V con n nodos, a los que se suman dos nodos adicionales: la fuente s y el sumidero t. Por lo tanto, el número total de vértices es:

$$|V(G)| = 2n + 2 = O(n).$$

- Respecto a las aristas:
  - Se agregan n aristas desde la fuente s a cada nodo de U.
  - Para cada par de contraseñas S_i y S_j (con $i < j$), se verifica la condición $S_i \subset S_j$. En el peor caso, se consideran hasta $O(n^2)$ pares. La verificación de inclusión tiene coste $O(d)$, y dado que d es constante ($d \leq 10$), se tiene un coste $O(1)$ por verificación.
  - Se agregan n aristas desde cada nodo de V al sumidero t.

  Por lo tanto, el número total de aristas es:

$$|E(G)| = O(n) + O(n^2) + O(n) = O(n^2).$$

2. Ejecución del algoritmo de flujo de costo mínimo:
El algoritmo de flujo de costo mínimo se implementa utilizando una variante que, en cada iteración, encuentra un camino aumentante de coste mínimo a través del método de Bellman–Ford.

- La complejidad de cada ejecución de Bellman–Ford es $O(|V| \cdot |E|)$. Dado que $|V| = O(n)$ y $|E| = O(n^2)$, cada iteración tiene coste:

$$O(n \cdot n^2) = O(n^3).$$

- El flujo total a enviar es, a lo sumo, n (ya que se envía una unidad de flujo por cada contraseña). Por lo tanto, en el peor caso se realizan $O(n)$ iteraciones.

La complejidad total de la fase de flujo es, por lo tanto:

$$O(n) \times O(n^3) = O(n^4).$$

3. Reconstrucción de la solución:
Una vez obtenido el matching óptimo a partir del flujo, se reconstruyen las cadenas y se genera la secuencia de pulsaciones. Esta fase consiste en recorrer los nodos y reconstruir las cadenas; en el peor caso se procesan $O(n)$ nodos con un coste $O(1)$ por cada uno, lo que implica una complejidad de $O(n)$.

Análisis global:
Sumando las complejidades de cada fase obtenemos:

$O(n^2)$ (construcción del grafo) + $O(n^4)$ (flujo de costo mínimo) + $O(n)$ (reconstrucción)

La complejidad dominante es $O(n^4)$.

Por lo tanto, la complejidad temporal global del algoritmo es $O(n^4)$ en el peor caso. □

Referencias

Pierre Charbit, Stéphan Thomassé, and Anders Yeo. The minimum feedback arc set problem is np-hard for tournaments. *Combinatorics, Probability and Computing*, 16:1 – 4, 2006. URL https://api.semanticscholar.org/CorpusID:36539840.